Machine Learning

based on *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman

Kanyes Thaker

Last updated: November 29, 2023

Note to the Reader

These notes provide a high-level summarization of Hastie et al's seminal text *The Elements of Statistical Learning*, which ranks among the most highly regarded classical machine learning references. These notes are not a comprehensive summary – rather they aim to distill information that is most widely applicable, most widely practical, or is most widely transferable to other fields of science and engineering. Certain topics which are more thoughtfully dealt with in dedicated texts (like Daphne Koller's text on PGMs or Ian Goodfellow's on neural networks, or even Bishop's text for a more comprehensive approach to Bayesian statistics) are left to those texts, and only briefly touched on here. The most attention is given to the fundamental chapters on linear regressors, linear classifiers, and model selection and evaluation, as per the authors' recommendation.

Contents

❖ Supervised Learning

The **machine learning** problem is one of interpreting patterns in data. A **model** (or *learner*) attempts to determine some output (either a continuous quantity (which we deem a **regression** problem) or a categorical variable (**classification**)) based on a set of available examples (**training data**), where each example has multiple different attributes (**features**) that the learner may learn from. If the examples come with a set of predefined outputs that the learner may observe, we say that the learner is **supervised** – if the learner must make sense of the data without these labeled outputs, it is **unsupervised**. Here we build a foundation on the supervised approach.

At a high level, the goal of the learning task is, given an input vector $\mathbf{x}$, make a good prediction of the output y, where we denote the prediction as $\hat{y} = f(\mathbf{x})$. We suppose that we have available to us a set of training measurements $(\mathbf{x}_i, y_i)$ where $\mathbf{x}_i$ is an example and y_i is the associated output label.

1.1 Least Squares and Nearest Neighbors

The simplest construction of f is that of a **linear model**, where we assume there is some linear relationship between the output and input, i.e. $\hat{y} = \beta_0 + \sum \mathbf{x}_i \beta_i$ (equivalently, $\hat{y} = \mathbf{x}^\top \beta$). The collection of $(\mathbf{x}, \hat{y})$ form an affine set in the $p + 1$ dimensional input-output space. We can then determine optimal values for β by minimizing the **squared error** (difference between prediction and target) – a method aptly dubbed **least squares**, which admits a closed-form solution through some simple calculus:

$$\beta^* = \arg\min_\beta \sum_{i=1}^{N} (y_i - \mathbf{x}_i^\top \beta)^2 = \arg\min_\beta \|\mathbf{y} - \mathbf{X}^\top \beta\|_2^2.$$

Ordinary Least Squares Solution

$$\beta^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

In the event that we wish to classify a point as one of two classes, we can draw a **decision boundary** that assigns class 1 to the point if $\mathbf{x}^\top \beta^* > 0.5$ and class 0 otherwise. The value that we make our decision on is the **decision threshold**.

In most cases, the data we have will not be cleanly **linearly separable**; the data in each class will overlap. Then we can make one of two assumptions:

1. The underlying distribution for each class is a single unimodal p-variate distribution with high variance. The errors are simply due to the variance in each class. Our linear assumption is good, and we can be confident that this is the "best we can do."

2. The underlying distribution for each class is actually a mixture of Gaussians, and the errors are not due to variance, but because the assumption of linearity we made earlier is actually incorrect.

The k**-nearest-neighbors** (kNN) method can help with the second scenario. In this scenario, we take an input $\mathbf{x}$, and find the k closest points $\mathbf{x}_i$ ("closest" meaning some metric, i.e. the Euclidean distance) in the training data, and average the corresponding k y_i's:

$$\hat{y} = \frac{1}{k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} y_i.$$

The linear decision boundary is smooth and stable, but relies on the assumption that a linear boundary is appropriate. It has low **variance** and high **bias**. On the other hand, kNN is highly unstable, but makes no assumptions; it has high **variance** and low **bias**. Variance can be thought of as a measure of "how much our model is influenced by the actual data" while bias can be thought of as a measure of "how much our model is influenced by simplifying assumptions." High-bias models tend to **underfit** and lose granularity while high-variance models tend to **overfit** and capture noise inherent in the data. All models have some complexity parameter that helps manage the **tradeoff** between bias and variance, and understanding this is key in designing machine learning systems.

These ideas presented so far can be encapsulated more broadly by **statistical decision theory**. Given an n-dimensional input vector $\mathbf{x}$ and a real-valued output y, we seek a function $f(\mathbf{x})$ (an **estimator**) that can approximate y. We do so by minimizing a **loss function** $L(Y, f(X))$ that quantifies how good our prediction is (here we substitute the random variables Y and X).

The minimizer of the expected square error $\mathbb{E}[(Y-c)^2]$ is $c = \mathbb{E}[Y]$. In our case, where we have additional information in the form of the data X, we can iterate the expectations and minimize $\mathbb{E}_X \mathbb{E}_{Y|X}[(Y - f(X))^2|X]$ pointwise, which gives us a final estimator of $f(X) = \mathbb{E}[Y|X]$. This is known as the **regression function**.

Both least squares and kNN approximate the conditional expectation (kNN does this directly, while least squares uses a linear **model**). kNN assumes that $f(\mathbf{x})$ can be approximated by a function that is locally constant, while least squares assumes that $f(\mathbf{x})$ is globally linear. While each assumption has its drawbacks, the majority of methods discussed here fall in the latter category.

The Curse of Dimensionality

We might intuit that nearest-neighbors methods are always optimal, since we can always collect a sufficient amount of data to find an approximate nearest neighbor.

This argument begins to fall apart in higher and higher dimensions, where we need exponentially more data in order to have a representative sample of all possible combinations of features. This problem is among a series of difficulties we encounter in high-dimensional spaces, and is commonly dubbed the **curse of dimensionality**. Using models with strict assumptions (like least squares) about the structure of f can help alleviate this problem. But these models fail horribly if the assumptions are wrong.

One common framing of the prediction problem is to represent the data as the model $y = f(\mathbf{x}) + \varepsilon$ where ε are zero-mean, i.i.d. values independent of $\mathbf{x}$. This **additive error** model closely resembles many real-world systems, where there is no strictly deterministic relationship between the output and input.

❖ Linear Models

A linear model assumes the regression function $\mathbb{E}[Y|X]$ is linear. These models are simple but incredibly powerful. Here we explore linear models with applications to regression (continuous output) and classification (discrete output).

2.1 Linear Regression

A linear model assumes that the target distribution is a linear function of its parameters, $y = \mathbf{x}^\top \beta$. As discussed before, the popular **ordinary least squares** (OLS) estimation method minimizes the residual sum-of-squares error $\sum(y_i - \mathbf{x}_i^\top \beta)$ over the training set $(\mathbf{X}, \mathbf{y})$. The estimator is the projection $\hat{\mathbf{y}}$ onto the subspace of $\mathbb{R}^n$ spanned by the columns of $\mathbf{X}$.

Solving the resulting optimization problem yields the closed-form **ordinary least squares** solution:

$$\hat{\beta} = \arg\min_\beta \|\mathbf{y} - \mathbf{X}\beta\|_2^2 = \arg\min_\beta \beta^\top \mathbf{X}^\top \mathbf{X}\beta - 2\mathbf{y}^\top \mathbf{X}\beta = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$$

The predicted values are $\hat{y} = \mathbf{X}\hat{\beta}$.

OLS admits multiple solutions if $\mathbf{X}$ is rank-deficient (**multicollinearity**), in which case we must eliminate columns or rows. Here are some other assumptions we make OLS relies on an assumption of **exogeneity**, wherein the error term is fully independent of $\mathbf{X}$; **homoskedasticity**, that all error terms ε have identical variance; there is no **autocorrelation** between error terms ($\text{Cov}(\varepsilon_i, \varepsilon_j) = 0$).

Suppose we have an estimate $\hat{\vartheta} = \mathbf{a}^\top \hat{\beta}$ using our OLS solution $\hat{\beta}$. $\hat{\vartheta}$ is an **unbiased estimator** since $\hat{\vartheta} = \mathbf{a}^\top \beta$. Using the triangle inequality, we can show that any other

unbiased estimator ϑ wil lhave $\text{Var}(\hat{\vartheta}) \leq \text{Var}(\vartheta)$ – this says that the OLS estimator is the unbiased estimator of least variance (this is the **Gauss-Markov theorem**).

In a multi-label case, where we expect each row in $\mathbf{X}$ to result in K possible predictions, we replace $\mathbf{y}$ in our least squares formulation with the $N \times K$ matrix $\mathbf{Y}$. Other than this, the formulation is similar – our loss function is then $\text{trace}[(\mathbf{Y}-\mathbf{XB})^\top(\mathbf{Y}-\mathbf{XB})]$. This yields the same solution of $\mathbf{B} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}$. This can be decomposed into $\mathbf{K}$ independent binary classifiers on each of the target classes.

2.2 Shrinkage

Least squares suffers from poor prediction accuracy (low bias, high variance), since there may be a lot of features (dimensions) that are difficult to interpret. We can attempt to remedy this problem by reducing the number of features that our model is learning. We can do this discretely, by selecting only the most influential features greedily (known as **subset methods**). Discrete methods, however, are prone to high variance – instead, we may opt for methods which will shrink the influence of features continuously, thereby reducing variability.

2.2.1 Ridge Regression

One popular shrinkage method is **ridge regression**, which shrinks our β coefficients towards zero by penalizing their magnitude (measured by their square, β_i^2, or for a vector $\|\beta\|_2^2$. In the land of neural networks this is known as **weight decay**. We might do this when we have highly correlated feature, where OLS might assign a hugely positive coefficient for one feature and balance it out with a hugely negative coefficient on its cousin. The ridge solution mitigates this by penalizing large-magnitude features more than small-magnitude features.

Ridge Regression

The ridge regression optimization problem is then

$$\hat{\beta} = \arg\max_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2 = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}.$$

Here λ is a parameter we set before we fit the model. Such a parameter that is manually set before the model is fit is known as a **hyperparameter**.

Since $\lambda > 0$, notice that even if $\mathbf{X}^\top\mathbf{X}$ is singular, $(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})$ must be nonsingular, the ridge regression minimizer is always unique (the objective function is *strictly convex*).

A **Bayesian** understanding of ridge regression models it as the mode of the posterior distribution with a Gaussian prior. If $\mathbf{y}|\mathbf{X}, \beta \sim \mathcal{N}(\mathbf{X}\beta, \sigma^2\mathbf{I})$, and we assume $\beta \sim \mathcal{N}(0, \tau^2)$

then

$$P(\beta|\mathbf{y},\mathbf{X}) \propto P(\beta)P(\mathbf{y}|\beta,\mathbf{X}) = \exp\left\{-\frac{1}{2\sigma^2}\left(\|\mathbf{y}-\mathbf{X}\beta\|_2^2 + \frac{\sigma^2}{\tau^2}\|\beta\|_2^2.\right)\right\}.$$

Maximizing this posterior (known as ***maximum a posteriori*** or **MAP**) – or equivalently, minimizing the negative log of the posterior – gives us the ridge regression solution.

Ridge regression can also be understood through the **singular value decomposition** $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$. For an $n \times m$ matrix $\mathbf{X}$ with rank p, $\mathbf{U}$ and $\mathbf{V}$ are $n \times p$ and $p \times m$ orthogonal matrices, where the columns of $\mathbf{U}$ span the column space of $\mathbf{X}$, and the columns of $\mathbf{V}$ span the row space. $\mathbf{D}$ is a $p \times p$ diagonal matrix whose entries are listed in order of descending magnitude, known as the **singular values**. Using the singular values,

$$\mathbf{X}\hat{\beta}^{ridge} = \mathbf{X}(X^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y} = \mathbf{U}\mathbf{D}(\mathbf{D}^2 + \lambda\mathbf{I})^{-1}\mathbf{D}\mathbf{U}^\top\mathbf{y} = \sum \mathbf{u}_j \frac{d_j^2}{d_j^2+\lambda}\mathbf{u}_j^\top\mathbf{y}.$$

Compare this to the OLS estimator, where $\mathbf{X}\hat{\beta}^{OLS} = \mathbf{U}\mathbf{U}^\top\mathbf{y}$ – we see that each coordinate of $\mathbf{y}$, represented in the basis $\mathbf{U}$, is shrunk by a factor of $d_j^2/(d_j^2+\lambda)$. This is significant, because the SVD is also associated with the **principal components** of $\mathbf{X}$. The sample covariance matrix is $\frac{1}{N}\mathbf{X}^\top\mathbf{X} \propto \mathbf{V}\mathbf{D}^2\mathbf{V}^\top$, the **eigendecomposition** of $\mathbf{X}^\top\mathbf{X}$. The eigenvectors $\mathbf{v}_j$ are then known as **principal components**, and since $\text{Var}(\mathbf{X}\mathbf{v}_i) = \frac{d_i^2}{N}$, the principal component $\mathbf{v}_1$ represents the direction of maximum variance (as do the rest of the $\mathbf{v}_i$). So all this means that $d_j^2/(d_j^2+\lambda)$ is smaller when d_j is smaller – meaning ridge regression shrinks weights around the directions of least variance the most. This is a reasonable assumption since we expect the longest directions to be the most informative.

2.2.2 Lasso

The lasso method is similar to the ridge regression formulation, but substitutes ℓ-2 norm penalty for an ℓ-1 norm one, so we attempt to find $\hat{\beta}^{lasso} = \arg\min_\beta \|\mathbf{y}-\mathbf{X}\beta\|_2^2 + \lambda\sum|\beta_j|$. The constraint is nonlinear in $\mathbf{y}$, and as a result lasso admits no closed-form solution, and its objective function is not differentiable everywhere due to the sharp corners of $|\beta_j|$. However, while ridge regression continuously shrinks all coefficients towards zero, lasso will set some coefficients as exactly zero, which induces **sparsity**. While we will not illustrate this here, this is equivalent to evaluating the posterior distribution with a Laplace distribution prior on β. We sometimes combine the ridge and lasso regularizers by linearly interpolating between them, i.e. $\text{argmin}_\beta \|\mathbf{y}-\mathbf{X}^\top\beta\|_2^2 + \lambda\|\beta\|_2^2 + (1-\lambda)\|\beta\|_1$, known as **elastic net**.

2.2.3 A Brief Comparison

Ridge regression performs a proportional weight shrinkage towards zero. Lasso shrinks coefficients by λ, truncating them at zero (**soft thresholding**). Best-subset selection selects only the top M largest coefficients (**hard thresholding**). Really, all these three

methods are Bayesian estimates with different priors – but all of them correspond to the maximum of the posterior distribution.

2.3 Linear Classification

The classification problem consists of generating a predictor $G(x)$ that outputs a member of a finite output set $\mathcal{G}$. We do this by splitting the input space into regions which define the output classes. In particular, we examine linear **decision boundaries** that define these regions.

One way to do this is to extend the idea in §1.1, where we train $O(k^2)$ linear-regression-based classifiers on each pair of classes, and classify a datapoint based on the class that gets the most votes from all the classifiers (**one-versus-one**). Alternatively we could train $O(k)$ binary classifiers where we select one class as the "positive" class and cast all others as "negative" (**one-versus-rest**). However, linear regression models are hard to interpret for classification tasks, since they don't neatly map to probabilities. Errors with these models are heteroscedastic.

A linear **discriminant function** δ_k is a linear function that returns a score for class k given a data point $\mathbf{x}$. A linear classifier would emit the class $\arg\max_k \delta_k$. Models that model the posterior probability $P(G = k|X)$ fall into this same category. More loosely, the decision boundary is linear if a monotone transformation (a **link function**) of δ_k/posterior probability is linear.

The **logit function** is an extremely important example. The logit or log-odds function $\text{logit}(p) : [0, 1] \to \mathbb{R} = \log(p/1-p)$ is a monotonic function that is easily interpretable as a ratio of probabilities. Formally, it is the inverse of the CDF of the logistic distribution. The next two examples (LDA and logistic regression) aim for linear logits. In the section on logistic regression we will see why the logit function in particular is useful over any other choice.

2.3.1 Linear Discriminant Analysis

We know that optimal classification is achieved through class posteriors $P(G|X)$. Let's model the class-conditional density as $P(X|G = k) = f_k(\mathbf{x})$ and the class prior probabilities as $P(G = k) = \pi_k$. We can estimate the full class prior from the law of total probability as $P(X) = \sum_k P(X, G = k) = \sum_k P(X|G = k)P(G = k) = \sum_k f_k(\mathbf{x})\pi_k$.

$$P(G = k|X = x) = \frac{f_k(\mathbf{x})\pi_k}{\sum_\ell f_\ell(\mathbf{x})\pi_\ell}.$$

Linear discriminant analysis (LDA) is the case where we model the class-conditional density as a multivariate Gaussian with shared covariance Σ:

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \mu_k)^\top \Sigma^{-1}(\mathbf{x} - \mu_k)\right\}.$$

So the log-ratio of the posteriors is a linear equation in $\mathbf{x}$:

$$\log \frac{P(G=k|\mathbf{X}=\mathbf{x})}{P(G=\ell|\mathbf{X}=\mathbf{x})} \propto \log \frac{\pi_k}{\pi_l} - \frac{1}{2}(\mu_k-\mu_\ell)^\top \Sigma^{-1}(\mu_k-\mu_\ell) + \mathbf{x}^\top \Sigma^{-1}(\mu_k-\mu_\ell),$$

The **linear discriminant functions** are then $\log f_k(\mathbf{x})$ (up to proportionality). So in LDA, we are trying to get linear log-odds by modeling the joint distribution $P(X, G=k)$ with $f_k(\mathbf{x})$, which we assume to be Gaussian.

$$\delta_k = \mathbf{x}^\top \Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k + \log \pi_k,$$

and $G(\mathbf{x}) = \text{argmax}_k\, \delta_k$. In practice, we do not know the parameters μ_k, Σ of the Gaussians, so we estimate them from our training data.

If the Σ_k are *not* all equivalent, our terms do not cancel as nicely, and we instead get **quadratic discriminant analysis** (QDA), with functions

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(\mathbf{x}-\mu_k)^\top \Sigma_k^{-1}(\mathbf{x}-\mu_k) + \log \pi_k.$$

QDA and LDA tend to provide similar estimates. QDA is more relaxed but suffers from an explosion in the number of parameters. LDA and QDA perform exceptionally well on a diverse slate of classification tasks, since Gaussian estimates are stable, and (even with high bias) maintain low variance. We can *regularize* QDA by shrinking the covariances to be somewhere closer to fully uniform (as in LDA). To do this, we use a regularized covariance matrix $\hat{\Sigma}_k(\alpha) = \alpha\hat{\Sigma}_k + (1-\alpha)\hat{\Sigma}$ where $\hat{\Sigma}$ is the "universal" LDA covariance.

To make computations easier, we can diagonalize $\Sigma_k = \mathbf{U}_k\mathbf{D}_k\mathbf{U}_k^T\,\mathbf{X}^*$. Then we can **sphere** the data as $\mathbf{X}^* = \mathbf{D}^{-1/2}\mathbf{U}^T\mathbf{X}$ (i.e. we center and scale the data so the covariance of the transformed data is $\mathbf{I}$). Then the discrimnant functions amount to finding the closest centroid to the data point in the sphered data space, mod π_k.

Since there are K centroids in the p-dimensional feature space, all centroids must lie in a $K-1$ affine subspace of $\mathbb{R}^p$. The closest centroid can be determined exclusively within this subspace; LDA is then a type of dimensionality reduction. To make separating the centroids easier, we then identify the top principal components to maximize the variance between them.

1. Compute the set of K centroids $\mathbf{M}$ $(K \times p)$ from the training data
2. Compute the shared within-class covariance $\mathbf{W}$
3. Compute $\mathbf{M}^* = \mathbf{M}\mathbf{W}^{-\frac{1}{2}}$, "sphering" $\mathbf{M}$
4. Compute $\mathbf{B}^*$, the covariance matrix of $\mathbf{M}^*$, where $\mathbf{B}$ is the between-class covariance. The eigenvectors $\mathbf{v}_\ell^*$ of $\mathbf{B}^*$ are then the coordinates of the optimal subspaces.

2.3.2 Logistic Regression

Here we examine a formulation which directly attempts to model the posterior distribution without the joint distribution. The general idea is this; we want to transform our linear model $\beta^\top \mathbf{x}$ into meaningful and interpretable probabilities. This transformation should be symmetric (class assignments are arbitrary), should be monotonic, have continuous derivatives, and saturate at $\pm\infty$. Additionally, the inverse transformation from probabilities *back* into a linear function should have some logical interpretation as a "decision boundary."

The logit function, along with its inverse the **logistic sigmoid** satisfy all these requirements. While they are not the only functions that do, their simple formulation have made them a wildly popular choice for this problem. The logit function is easily interpretable as a ratio of probabilities (intuitively mapping to the concept of a "decision boundary"). The logistic sigmoid function is the CDF of the logistic distribution; it is symmetric about zero, has continuous derivatives, and (being a CDF) necessarily has a codomain of $[0, 1]$ and saturates at $\pm\infty$. An alternative combination would be the **probit** function and Gaussian CDF Φ, but the probit function is much less interpretable as a ratio of probabilities.

The **logistic regression** model consists of $K - 1$ logit transformations of form

$$\log \frac{P(G = k|X = x)}{P(G = K|X = x)} = \beta_k^\top \mathbf{x}.$$

Based on the above discussion, individual posteriors can be written as follows. You can confirm that these functional forms behave as desired.

$$P(G = k|X = x) = \frac{\exp(\beta_k^\top \mathbf{x})}{1 + \sum_\ell \exp(\beta_\ell^\top \mathbf{x})} \quad P(G = K|X = x) = \frac{1}{1 + \sum_\ell \exp(\beta_\ell^\top \mathbf{x})}$$

We will use $P(G = k|X = x) = p_k(x; \vartheta)$ as shorthand.

We normally fit the logistic regression model using the maximum (log) likelihood, with the conditional likelihood of G given X $P(G|X)$. In the binary case, $p_0(x; \vartheta) = 1 - p_1(x; \vartheta)$, and the log-likelihood can be written as

Binary Cross-Entropy Loss

$$\ell(\beta) = \sum_{i=1}^{N} y_i \log p(x_i; \beta) + (1 - y_i) \log(1 - p(x_i; \beta))$$

Maximizing the log-likelihood amounts to minimizing the negative log-likelihood. Since the negative log-likelihood takes the form of the **cross-entropy** between two distributions $H(p; q) = -\sum p \log q$, we call this the **binary cross-entropy loss**. As usual, we can set

the derivative of the loss to zero to find the optimum.

$$\ell(\beta) = \sum_{i=1}^{N} y_i \beta^\top \mathbf{x}_i - \log(1 + e^{\beta^\top \mathbf{x}_i}) \implies \frac{\partial \ell(\beta)}{\partial \beta} = \sum \mathbf{x}_i (y_i - p(\mathbf{x}_i; \vartheta)) = \mathbf{X}^\top (\mathbf{y} - \mathbf{p}),$$

which is an equation nonlinear in β. Since there is no closed-form solution, we may use a numerical method (such as Newton's method) to find the zeros:

$$\beta^{new} \leftarrow \beta^{old} - \left(\frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^\top} \right)^{-1} \frac{\partial \ell(\beta)}{\partial \beta}, \quad \frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^\top} = \sum \mathbf{x}_i \mathbf{x}_i^\top p(\mathbf{x}_i; \beta)(1 - p(\mathbf{x}_i; \beta)) = -\mathbf{X}^\top \mathbf{W} \mathbf{X}$$

where $\mathbf{W}$ is a diagonal matrix with elements $p(\mathbf{x}_i; \beta)(1 - p(\mathbf{x}_i; \beta))$. The update rule can be simplified as

$$\beta^{new} \leftarrow \beta^{old} + (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top (\mathbf{y} - \mathbf{p}) = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W} (\mathbf{X} \beta^{old} + \mathbf{W}^{-1} (\mathbf{y} - \mathbf{p})) = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W} \mathbf{z}$$

This variant of least squares, where there is a weighting parameter that is modified on each iteration, is called **iteratively reweighted least squares (IRLS)**.

2.3.3 A Brief Comparison

So when do we use LDA versus logistic regression? Both LDA and logistic regression yield log-odds linear in $\mathbf{x}$. However, they estimate their coefficients differently. Logistic regression estimates $P(G|X)$ without making any assumptions about $P(X)$, instead maximizing the **conditional likelihoods** $P(G = k|X)$. Meanwhile, LDA maximizes the full log-likelihood using the joint density $P(X, G = k) = \varphi(X; \mu_k, \Sigma)\pi_k$. This also involves the marginal density $P(X)$, which we assume to be a Gaussian mixture.

The Gaussian assumption in LDA allows us to estimate parameters with lower variance; if the data is actually Gaussian, modeling the joint density (a **generative model**) requires 30% less data for the same performance as just modeling the conditional density (**discriminative**).

However, outliers (which logistic regression down-weights) will contribute to the common covariance matrix used in LDA. Forcing us to use this assumption on the marginal distribution means that LDA is always well-defined where logistic regression may not be, such as in a case where the data is completely linearly separable. In practice, logistic regression is typically a safer and more robust classifier than LDA, as it requires fewer assumptions – but these models generally perform very similarly.

2.3.4 Separating Hyperplanes

Separating hyperplane classifiers explicitly try to split the data into different classes as well as possible, making no attempt to model the actual data. Classifiers that compute linear combinations of input features and return a sign $+/-$ are called **perceptrons**.

The Rosenblatt **perceptron learning algorithm** tries to find a separating hyperplane by minimizing the distance of misclassified points to that boundary. It aims to minimize the error incurred by a misclassified response:

$$D(\beta) = -\sum_{i \in \mathcal{M}} y_i(\mathbf{x}_i^\top \beta)$$

where $\mathcal{M}$ is the set of misclassified points. Correctly classified points incur no loss. This loss function has continuous (linear) gradients, as $\nabla_\beta D(\beta) = -\sum_{i \in \mathcal{M}} y_i \mathbf{x}_i$. This piecewise-linear function is minimized using **stochastic gradient descent**, meaning instead of computing the sum of gradients over all observations and then taking a step, we take a step after each individual observation. If the classes are linearly separable, this process is guaranteed to converge in a finite number of steps, but there may be infinitely many solutions depending on the initial conditions. For non-linearly separable data, the algorithm will not converge.

❖ Basis Expansions and Regularization

It is unlikely that the true relationship between $\mathbf{X}$ and $\mathbf{Y}$ is linear. This section moves beyond linearity. In particular, we can introduce a series of M transforms of $\mathbf{X}$ (called **basis functions**), denoted $h_m(\mathbf{X})$, and fit a linear model in the transformed space, i.e. $\hat{f}(\mathbf{X}) = \sum_{m=1}^{M} \beta_m h_m(\mathbf{X})$. For example, we can have $h_{i,j}(\mathbf{X}) = \mathbf{X}_i \mathbf{X}_j$ as a way to model a quadratic relationship between $\mathbf{X}$ and $\mathbf{y}$. We can likewise use the logarithm, square root, or any other number of basis functions.

In particular, polynomial basis functions are valuable because they can locally approximate many functions via the Taylor series. However, the number of basis functions explodes in the degree of the polynomial, so we typically split the domain into discrete intervals and fit separate low-degree polynomials to each interval.

Naturally, there will be discontinuities at the boundaries (**knots**) between the intervals. To address this, we might want to add continuity conditions at the knots. In particular, we want continuity up until the 2nd derivative, the point at which the human eye will not notice the knot locations. As such, we will often fit cubic functions, known as **cubic splines**. For the second derivative in particular, we can control variance around the knot locations by ensuring that the second derivative is zero at knot points (**natural** cubic splines).

We can determine the optimal knot locations by finding the interpolating function that minimizes the penalized residual sum-of-squares:

$$RSS(f, \lambda) = \sum_{i=1}^{N} (y_i - f(x_i)^2) + \lambda \int f''(t)^2 dt,$$

where $\lambda = 0$ allows any function to be used while $\lambda = \infty$ corresponds to simple least squares (since no curvature can be tolerated). This is known as **smoothing spline** interpolation. For a dataset with N terms, the above regularizer has a solution corresponding to a natural cubic spline with knots at all unique values of $\mathbf{x}$. While this may seem too loose, with $O(N)$ parameters, the penalty term λ shrinks the overall function towards linear.

❖ Kernel Methods

The general idea of this section is to estimate the regression function $f(\mathbf{X})$ by making a local estimation using a simple model for each query point $\mathbf{x}_0$. This local model is fitted using a neighborhood of points close to $\mathbf{x}_0$, where points further from $\mathbf{x}_0$ are weighted based on a weighting function (a **kernel**) $K_\lambda(\mathbf{x}_0, \mathbf{x}_i)$. The parameter λ is the only one determined from the training set, and denotes the width of the neighborhood (the kernel's **bandwidth**). A simple example is the $k-$nearest neighbors algorithm, where a query point receives a value equal to the average of the k closest points in squared distance.

The k-nearest neighbors average function is piecewise-constant and hence discontinuous. Alternatively, we can use a kernel function to assign weights that decay as we stray further from the target point, and then factor that kernel function into our average. A popular choice is the following:

Nadaraya-Watson Kernel-weighted Average

$$\hat{f}(\mathbf{x}_0) = \frac{\sum K_\lambda(\mathbf{x}_0, \mathbf{x}_i) y_i}{\sum K_\lambda(\mathbf{x}_0, \mathbf{x}_i)}, \quad K_\lambda(\mathbf{x}_0, \mathbf{x}) = D\left(\frac{|\mathbf{x} - \mathbf{x}_0|}{h_\lambda(\mathbf{x}_0)}\right).$$

The function h_λ is a window function that changes the width of the kernel depending on $\mathbf{x}_0$. When h_λ is constant, we say we have a *metric* window width. Methods where the bandwidth is not fixed are known as **adaptive** or **variable** kernel methods, and can be quite powerful. For nearest-neighbor windows, where the bandwidth depends on the density of points in the neighborhood, we would like a kernel with compact support. The Epanechikov kernel is an oft-used kernel with compact support; the Gaussian density kernel is a commonly used kernel with non-compact support.

So with our new kernel-weighted average we have avoided the discontinuities of the nearest-neighbors moving average – however, these methods issues at boundaries where our "neighborhoods" are less well-behaved; this can introduce bias to our model. Instead of simply taking an average of the points in the neighborhood, we can instead fit a **locally weighted linear regression** to the points in the neighborhood, where the weights are the kernel weights as before. The combined weights of the kernel and regression are

known as the **equivalent kernel**. Local linear regression automatically modifies the kernel to correct for bias up to the first order – **automatic kernel carpentry**. We can also keep going, and fit higher and higher order polynomials locally; but linear fits are most reliable due to their behavior at boundaries.

4.1 Kernel Density Estimation

Kernel density estimation is an older *unsupervised* technique that lends itself well to local *classification*. As before, the local estimate is done via an average within a metric neighborhood, a la $\hat{f}_{\mathbf{X}}(\mathbf{x}_0) = \frac{\#\mathbf{x}_i \in \mathcal{N}(\mathbf{x}_0)}{N\lambda}$. As before, we typically add a weighting kernel to retrieve what is known as the **Parzen estimate**. the Parzen window usually uses some probability distribution $\varphi_\lambda(\mathbf{x}_i - \mathbf{x}_0)$ as the kernel function, centered at $\mathbf{x}_i - \mathbf{x}_0$ parameterized by λ. A common choice is the Gaussian kernel, in which λ represents the standard deviation:

$$\hat{f}_{\mathbf{X}}(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\varphi_\lambda(\mathbf{x}_i - \mathbf{x}_0) = \frac{1}{N(2\lambda^2\pi)^{p/2}}\sum_{i=1}^{N} e^{-\frac{1}{2}(\|\mathbf{x}_i - \mathbf{x}_0\|/\lambda)^2}.$$

We can use this for classification, for class priors π_j, by using Bayes' theorem:

$$\hat{P}(G = j|X = x_0) = \frac{\hat{\pi}_j \hat{f}_j(\mathbf{x}_0)}{\sum \hat{\pi}_k \hat{f}_k(\mathbf{x}_0)}.$$

4.1.1 Naive Bayes Classifier

The **naive Bayes** model is good in high dimensional spaces where modeling the density directly is inefficient. Naive Bayes assumes that, for a class $G = j$, the features $\mathbf{X}_k$ are independent, so

$$f_j(\mathbf{X}) = \prod_{k=1}^{p} f_{jk}(\mathbf{X}_k).$$

The individual class-conditional marginal densities $f_{jk}(\mathbf{X}_k)$ can be estimated independently using one-dimensional kernel density estimates, such as the Gaussian Parzen window estimate above. This is a rather strong assumption, but the resultant classifier performs suprisingly well, usually because even if the individual class densities are heavily biased, the posterior may not be. We can derive the decision boundary via the logit transform, so

$$\log\frac{P(G = \ell|X)}{P(G = J|X)} = \frac{\pi_\ell f_\ell(\mathbf{X})}{\pi_J f_J(\mathbf{X})} = \log\frac{\pi_\ell}{\pi_J} + \sum_{k=1}^{p}\frac{f_{\ell k}(\mathbf{X}_k)}{f_{Jk}(\mathbf{X}_k)}.$$

4.1.2 Radial Basis Functions

In the previous section we discussed modeling non-linear relationships through basis functions, some of which are defined locally (such as in the case of splines). Kernel

methods are flexible because the fit simple models in a region local to $\mathbf{x}_0$. **Radial basis functions** combine these ideas, where the kernel functions $K_\lambda(\xi, \mathbf{x})$ are treated as basis functions, indexed by a scale parameter λ and a location (*prototype*) parameter ξ. Then our fitting function becomes

$$f(\mathbf{X}) = \sum K_{\lambda_j}(\xi_j, \mathbf{x})\beta_j.$$

We commonly use the Gaussian kernel here, again with ξ_j as the mean and λ_j as the standard deviation. We can learn these parameters, along with the weights β_j, with linear regression, minimizing over all three; this is known as an **RBF network**, an alternative to sigmoidal neural networks; it can be solved through nonconvex optimization, similar to neural networks. We could alternatively fit the kernel parameters ξ, λ first, and then optimize β separately, by attempting to model the data $\mathbf{X}$ with a Gaussian mixture directly. While we may also consider simplifying by holding $\lambda = \lambda_j$ fixed, this might lead to regions of little support (*holes*).

❖ Model Selection

In practice, we evaluate machine learning models based on their ability to **generalize** to new or unseen data. This section discusses ways of evaluating and interpreting generalization.

5.1 Bias and Variance

Assume some model minimizing a loss function $L(\mathbf{Y}, \hat{f}(\mathbf{X}))$. For a test set of data points denoted $(\mathbf{X}, \mathbf{Y})$, and a model trained over a training set τ, the **test error** is the average of the loss function evaluated at each point in the test set, $Err_\tau = \mathbb{E}[L(\mathbf{Y}, \hat{f}(\mathbf{X})|\tau]$. A related quantity is the expected test error $\mathbb{E}[Err_\tau]$, the average over all training sets. The **training error** is the average loss over the training sample itself.

As our model increases in complexity, it is more able to fit to complex underlying structures in the training data (variance increases). Simpler models make more simplifying assumptions (bias increases). There is some intermediate in the middle that is optimal. Unfortunately, training error is generally not a good estimate of test error, since increasing variance will always decrease the error on the training set, but may increase error on the test set.

In general, there are two separate goals when it comes to estimating test error. The first is **model selection**, the practice of comparing different models to find the best one. The second is **model assessment** – having selected a best model, accurately estimating its generalization powers on new data. With large datasets, a good pratice is to **split** the data into uniformly partitions. A **training set** will be used to train each model, a **validation set** will be used for model selection, and a **test set** will be used for model

assessment. In this section we primarily explore situations where it may not be possible to dedicate large sections of data entirely for validation or testing.

Recall that we often assume that $\mathbf{Y} = f(\mathbf{X}) + \varepsilon$, i.e. some pure function of the data plus some irreducible error. For some fitting function $\hat{f}(\mathbf{X})$, we then have the following:

Bias-Variance Decomposition

$$Err(\mathbf{x}_0) = \mathbb{E}[(\mathbf{Y} - \hat{f}(\mathbf{x}_0))^2 | \mathbf{X} = \mathbf{x}_0] = \varepsilon^2 + [\mathbb{E}\hat{f}(\mathbf{x}_0) - f(\mathbf{x}_0)]^2 + \mathbb{E}[\hat{f}(\mathbf{x}_0) - \mathbb{E}\hat{f}(\mathbf{x}_0)]^2$$
$$= \varepsilon^2 + \text{Bias}(\hat{f}(\mathbf{x}_0))^2 + \text{Var}(\hat{f}(\mathbf{x}_0)).$$

The bias measures how much the average of our estimate differs from the true mean, and the variance is the expected squared deviation of the estimate about its mean. The bias term itself can also be broken down into two quantities, the *model bias* (error between the true function and the best linear approximation) and *estimation bias* (error between the average estimate and the best linear approximation). In the OLS case, the estimation bias is zero, and in regularized case the estimation bias is positive, and we trade it off for decreased variance. In cases where the decrease in variance is greater than the square of the estimation bias, the regularized fit will yield a lower overall error.

5.2 Bayesian Information Criterion

The training error is typically less than the test error. Remember that we typically think of $\mathbf{y}$ as being random, as $f(\mathbf{X}) + \varepsilon$. In addition to the training error, which captures the error for the current training set, it's also interesting to consider the *average* training error over all possible values of $\mathbf{y}$ for our training set. This value is known as the **in-sample error**. Since our model was fit to the $\mathbf{y}$-values we observe, we naturally expect the training error to be less than the in-sample error. The difference between these two is known as the **optimism** – a measure of how much influence the target values we observe have on the model, versus the underlying distribution that $\mathbf{y}$ belongs to. In other words, it is a measure of how optimistic we are that our sampled training set will capture the true data distribution. The average optimism over all training sets is denoted ω; the optimism is proportional to $\text{Cov}(\hat{\mathbf{y}}, \mathbf{y})$, where the proportionality constant depends on the number of parameters (number of basis functions) d.

We rarely use the in-sample error in practice, and most modern literature considers the training error and in-sample error interchangeable (which isn't exactly true). Rather, we might use some estimate instead. The **Akaike information criterion (AIC)** does precisely this – the expected log likelihood penalized by the number of parameters:

$$AIC = -2\mathbb{E}[\log P_{\hat{\vartheta}}(Y)] = -\frac{2}{N}\mathbb{E}\left[\sum \log P_{\hat{\vartheta}}(y_i)\right] + 2\frac{d}{N} = -\frac{2}{N}\mathbb{E}[\hat{\mathcal{L}}] + 2\frac{d}{N}.$$

The objective is that the AIC curve can simulate the test error curve, so minimizing the AIC will be a good way to find the number of parameters that would also minimize the test error.

The **Bayesian information criterion** (BIC) is also applicable when the fitting is done through maximizing a log-likelihood.

Bayesian Information Criterion (Schwarz Criterion)

$$BIC = -2\hat{\mathcal{L}} + (\log N)d$$

The BIC is proportional to the AIC where the factor of 2 is replaced with $\log N$. As a result, BIC tends to penalize complex models more heavily. The BIC, however, is motivated quite differently. The BIC represents the posterior probability distribution $P(\mathcal{M}|\mathbf{Z})$, where $\mathbf{Z}$ is a training set and $\mathcal{M}$ is a candidate model. For a model $\mathcal{M}_m$ with prior $P(\vartheta_m|\mathcal{M}_m)$, the posterior is

$$P(\mathcal{M}_m|\mathbf{Z}) = \int P(\mathbf{Z}|\vartheta_m, \mathcal{M}_m)P(\vartheta_m|\mathcal{M}_m)d\vartheta_m$$

(we typically assume the model prior is uniform). The posterior odds, $P(\mathcal{M}_m|\mathbf{Z})/P(\mathcal{M}_n|\mathbf{Z})$ can then be evaluated just from the ratio of the likelihoods $P(\mathbf{Z}|\mathcal{M}_m)/P(\mathbf{Z}|\mathcal{M}_n)$. The BIC follows from an estimation of $P(\mathbf{Z}|\mathcal{M}_m)$ (details to important). The BIC can also be formulated from an optimal coding viewpoint, in which case it is called the **minimum description length** (MDL).

5.2.1 Vapnik-Chervonekis Dimension

The Vapnik-Chervonekis dimension (VC) gives a general measure of complexity for a set of points, along with a bound on the optimism. The VC dimension of a class of functions is the largest number of points that can be **shattered** by the members of the class. A set of points can be shattered by a function if, no matter how we assign binary labels to those points, the function can separate them. For example, a line in the plane can always separate 3 points into two classes, no matter how the points are labeled – but this is not true for 4 points, so the VC dimension of straight lines in the plane is 3. The VC dimension is useful in producing probabilistic bounds for the test error.

5.3 Cross-Validation

The simplest method for estimating error is cross-validation. In an ideal case, we could set aside a validation set for our data and use it to assess model performance, and thereby calculate the error directly. Since data is often scarce, we can try to split the data into K equal-sized parts. We can then train each model K times, training on $K-1$ parts and validating on the one outstanding. Our prediction error is then the mean of the

individual errors for each training iteration. This is known as ***k*-fold cross validation**. $K = 5$ and $K = 10$ are commonly used values since they strike a good compromise between bias and variance.

5.4 Bootstrap

The bootstrap is another method for estimating the expected prediction error. The general idea is to train a model over a training set $\mathbf{Z}$, and then generate B randomly sampled (*with replacement*) subsets from the training data with the same size as the training set. We then can fit our model to each sample and estimate our error based on how each sample's model approximates the original. However, this is unstable, since we are treating the original model as a test set, and the bootstrapped models share some training data with the original model (leakage). Typically we will run some kind of cross-validation on the bootstrapped samples, so that we only evaluate the bootstrap predictions for a given data point using samples that did not contain that data point. Bootstrap models can also be used for prediction; the act of creating a model by averaging the predictions of several boostrapped models is called **bagging**.

❖ Model Inference and EM

Most methods so far have fit models by minimizing a loss function via maximum likelihood. This section generalizes the maximum likelihood approach.

Bootstrap methods in general use an ensemble of small models to simulate a large model. Consider a fitting problem where our dataset is one-dimensional. We might use basis functions or spline interpolation to fit a nonlinear model to this data with least squares, as we have in the past. We could then construct a confidence interval based on the mean error at each point. Alternatively, we could fit numerous (say, 200) fitting functions over smaller subsets of the data using the bootstrap method. Then the 95% confidence interval can be determined by finding the 5th smallest and 5th largest prediction at each datapoint. These two methods yield surprisingly similar results.

We call the above idea **nonparametric** bootstrap because it uses the raw data to generate the dataset for the bootstrapped model. This is in contrast to **parametric** boostrap, where we generate the bootstrap training set by adding noise on top of the predicted values. As the number of bootstrap samples goes to infinity, the boostrap confidence interval will approach the least squares confidence intervals, so long as the model errors are additive Gaussian. The parametric bootstrap does not agree with least squares in general – rather, it agrees with maximum likelihood.

Recall that the maximum likelihood method typiccally assumes a probability function $g_\vartheta(\mathbf{z}_i)$ for our data that is parameterized by some ϑ (this is **parametric** maximum likelihood). The likelihood function is the product of $g_\vartheta(\mathbf{z}_i)$ for all data points. For

example, if we assume g is Gaussian, then $\vartheta = (\mu, \sigma^2)$. The log likelihood is the sum of $\log g$ for each $\mathbf{z}_i$. The bootstrap can be thought of as the computer implementation of maximum likelihood. It lets us compute maximum likelihood estimates and error bars when the formulas themselves may not be available.

The maximum likelihood approach takes a frequentist approach, wherein we take the values we observe in the data at face value. As a result, they can be brittle – a maximum likelihood model for fair coin flips that has a dataset of 3 heads will predict heads ever after. Bayesian approaches attempt to encode some prior knowledge about the world and allow the data to update the prior into a posterior distribution $P(\vartheta|\mathbf{Z})$. While maximum likelihood estimates use the optimal $\hat{\vartheta}$ to compute predictions, $P(\mathbf{z}_{new}|\hat{\vartheta})$, the Bayesian approach uses the posterior:

$$P(\mathbf{z}_{new}|\mathbf{Z}) = \int P(\mathbf{z}_{new}|\vartheta)P(\vartheta|\mathbf{Z})d\vartheta.$$

6.1 Expectation-Maximization

Consider a dataset with two peaks (bimodal). A single Gaussian cannot be used to learn such data. Instead we might choose to use two Gaussians in a mixture model, where the two distributions Y_1 and Y_2 are mixed as $Y = (1 - \Delta)Y_1 + \Delta Y_2$. This is a **generative** representation of the mixture.

It would be very difficult to directly fit such a model using log likelihood – we end up with a tricky sum inside the log term. Instead, we could consider using our *latent* (hidden) variables Δ. If $\Delta_i = 1$ then y_i comes from the second model, and if $\Delta_i = 0$ then y_i comes from the first. If we knew all the Δ_i's, we could compute maximum likelihood estimates for models 1 and 2 by exclusively taking those data with $\Delta_i = 0$ or 1, respectively.

Since we don't know the actual values of Δ_i, we can instead just use the expected value $\gamma_i = \mathbb{E}[\Delta_i|\vartheta, \mathbf{Z}] = P(\Delta_i = 1|\vartheta, \mathbf{Z})$, known as the **responsibility**. Then we have an algorithm that loops in two steps. The **expectation** step assigns responsibilities based on the current estimates of the parameters of the model. The **maximization** step uses those responsibilities to then recompute the model parameters. This is the **expectation-maximization (EM)** algorithm. We repeat these steps until convergence. The above is an illustrative example – the **generalized** EM algorithm expands the procedure to a more general space of latent variables, but the overall idea remains the same. In the GEM algorithm, we augment our observed data $\mathbf{Z}$ with *latent* (unobserved) data $\mathbf{Z}^m$. We can then treat this as a joint maximization problem.

6.2 Gibbs Sampling

Either through direct MAP modeling or the EM algorithm, we have a Bayesian model. Now we would like to sample from the posterior so that we may make inferences about its parameters. In general, it is difficult to compute the integral for the posterior directly. We will often use a **Markov chain Monte Carlo** (MCMC) approach to posterior sampling – in particular, we explore the Gibbs sampling procedure, closely related to EM.

In general, suppose we have k random variables whose joint distribution we wish to sample. Often times it is difficult to model the joint distribution directly, but it can be simple to sample from the conditional distributions $P(U_j|U_1, ..., U_{j-1}, U_{j+1}, ..., U_k)$. The **Gibbs sampling** procedure draws a sample from each of these distributions, for each $1 \leq j \leq k$, repeatedly until the distribution stabilizes, at which point we have a sample from the joint distribution. The conditional distribution is usually simple enough to where we can use more traditional sampling techniques.

Gibbs Sampling

1. Select some initial values $U_k^{(0)}$ for each value of k.
2. Until convergence:
   (a) For each $1 \leq j \leq k$, generate $U_k^{(t)}$ by sampling from

$$P(U_j^{(t)}|U_1^{(t)}, ..., U_{j-1}^{(t)}, U_{j+1}^{(t-1)}, ..., U_k^{(t-1)})$$

So long as the Markov chain is ergodic, its stationary distribution is exactly the joint distribution – not a surprise, since the marginal densities are never touched in this process. In cases where we have some variables that are very highly correlated, our Markov chain might get stuck in one region of state space and fail to explore the whole space in any reasonable number of iterations. In cases like this, other algorithms like the Metropolis-Hastings algorithm or Hamiltonian Monte Carlo might be preferred.

We can think of Gibbs Sampling as similar to the EM algorithm, where we consider the latent data $\mathbf{Z}^m$ as an additional parameter for the Gibbs sampler.

❖ Additive Models

Previously we mentioned that we can use predefined basis functions to model non-linear relationships in our data. We can add more flexibility to this idea by exploring **generalized additive models**:

$$\mathbb{E}[Y|X_1, X_2, ..., X_p = \alpha + \sum_{i=1}^{p} f_i(X_i),$$

where the f in question are nonparametric smooth functions.

Tree-based models are those that partition the data space into hypercubes recursively (usually with binary splits) and then independently fit a predictor to the data within each hypercube. To run inference on a tree-based model, we just need to know what region in the data space the inference-time point lies in, and use the model fit to that region. These models are popular in the medical sciences, since they mimic how a doctor might think about a problem, as a series of decisions with a different model for each decision. The locations of the boundaries of the hypercubes (*splitting variables*) are determined greedily, as finding the optimum partition is computationally infeasible. The size of the tree is a hyperparmeter that determines the model's complexity – a tree too big will overfit, and a tree too small may underfit. Typically, we use a **pruning** process to decide how big a tree should be. We grow out a rather large tree, and then collapse nodes of the tree (hypercubes) to form a subtree that optimizes a cost function (known as **cost-complexity pruning**) by collapsing (combining) internal (non-terminal) nodes. The cost function in question is known as a **node impurity measure**, i.e. it measures how polluted the children of the split node are. The more non-homogenous the child, the higher the measure. In classification tasks, common impurity measures are the **Gini index** and **cross-entropy**, two differentiable functions which peak when the child node has mixed-up classes and continuously decay as the classes get more and more homogenous.

Tree-based regression and classification models are often intuituive, easy to visualize, and simple to implement. Each individual model in the tree might be very simple (the **classification and regression tree** algorithm, or **CART**, fits constant functions to each terminal node in the tree) which is easier than trying to fit a complex non-linear model all-at-once, and the overall model will still be a good estimate of the non-linearity. However, they do suffer from instability (small changes in the data may result in drastically different splits), a non-smooth decision surface, and can be overall expensive to train as the number of individual models might be very high.

7.1 Hierarchical Mixture of Experts (HME)

While tree-based methods like CART have hard-line split boundaries, the HME procedure has soft probabilistic boundaries. This property, along with the choice to fit logistic/linear regression models to each node versus constant functions, will result in a smooth optimization problem, and is formally a type of mixture model. It can be thought of as a tree with soft splits. In literature, the terminal nodes (the regions where we fit a regression/classification function) are called **experts**, and the non-terminal nodes are **gating functions** (determine probabilities to assign to the subtrees), hence the name "hierarchical mixture of experts." Each expert fits some function $Y \sim P(y|x, \vartheta_{j\ell})$,

and each gating function outputs a probability

$$g_j(x, \gamma_j) = \frac{e^{\gamma_j^\top x}}{\sum e^{\gamma_k^\top x}}$$

for all j children in the split. The most convenient way to learn the mixture probabilities (i.e. the gating functions) is to use the EM algorithm, as discussed earlier. While HME can offer better, more representative performance compared to CART, its probabilistic nature makes it impossible to find an optimum splitting topology, and its nature as a mixture model makes fitting it much more expensive as well. HME also lacks a lot of the interpretability that CART offers.

❖ Boosting

Boosting is the concept that an ensemble of weak classifiers will result in a more powerful "committee." While this may sound similar to other committee-based techniques like bagging, where we average multiple bootstrapped models, boosting is a fundamentally different technique.

Consider a two-class problem whose outputs are encoded as -1 or 1. The in-sample error is then $\frac{1}{N}\sum \mathbf{1}(y_i \neq G(\mathbf{x}_i))$, and the expected generalization error is $\mathbb{E}_{XY} I(Y \neq G(X))$. The general idea is to begin with a very weak classifier, whose outputs are only slightly better than random. We train m weak classifiers, and between each iteration, we weight misclassified training samples more heavily for the next classifier, so the next weak classifier in the sequence must concentrate on training examples missed by earlier classifiers. Our final output is then a weighted majority vote of all the candidates, where the weights depend on the error rates of each classifier ($\log(1 - err_m/err_m)$). This algorithm is known as **AdaBoost.M1**. The weak learners in question are often **decision tree stumps** – a classification tree with two nodes. A decision stump is a classifier which picks one feature, and determines a threshold for that feature which best splits the data.

All in all, we are trying to minimze a general loss function for general basis functions:

$$\min_{\alpha,\beta} L\left(y, \sum \alpha_n f_n(\mathbf{x}, \beta_n)\right)$$

for some mixing coefficients α and basis functions f parameterized by β. The sum inside of the loss function is difficult to deal with, so we may opt to deal with it iteratively. In the AdaBoost case, the individual decision stumps G_m are the basis functions, weighted by the expansion coefficients α. Boosting corresponds to a technique known as **foreward stagewise additive modeling**, where we estimate each f_m fully sequentially, without changing previous estimates.

$$f_m(\mathbf{x}) = f_{m-1} + \alpha_m b(\mathbf{x}, \beta_m), \quad (\alpha_m, \beta_m) = \arg\min_{\alpha,\beta} \sum L(y_i, f_{m-1}(\mathbf{x}_i) + \alpha b(\mathbf{x}_i; \beta))$$

where α are the expansion coefficients, b is a basis function and β are the parameters.

AdaBoost is equivalent to forward stagewise additive modeling where the loss function in question is $L(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x}))$. AdaBoost was not designed with this relationship in mind, but the nature of the exponential loss function gives it a nice analogy to the log-odds of $P(Y = 1|x)$.

8.1 Boosting Trees

We previously mentioned tree-based methods, where we partition the data space into hypercubes and fit constant functions to each hypercube. The actual partitioning scheme is intractable and optimized greedily. The **boosted tree model** is the sum of classification trees, induced in the forward-stagewise manner mentioned above. If we let Θ denote the set of parameters R_j, γ_j, where R denotes the region and γ is the mean of points in that region, the boosted tree model becomes:

$$f_M(\mathbf{x}) = \sum T(\mathbf{x}; \Theta_m), \quad \hat{\Theta}_m = \arg\min_m \sum L(y_i, f_{m-1}(\mathbf{x}_i) + T(\mathbf{x}_i; \Theta_m)).$$

Finding the regions R_j is even more difficult in this case than in the normal tree case. For simple cases – such as binary classification with exponential loss – the problem becomes noticeably easier (no more difficult than the CART case), and we can use the AdaBoost method for boosting classification trees. Other loss functions may give more robust classifiers but are not simple to solve.

8.2 Gradient Boosting

AdaBoost happens to correspond to the forward stagewise additive model with exponential loss. Other loss functions can be used as well – however, they are less trivial to compute, and as such we must result to gradient methods to optimize each iteration. The method of optimizing a forward stagewise additive model with an arbitrary loss function through gradient descent is known as **gradient boosting**. The gradient boosting algorithm begins with some constant model f_0, fits a weak learner h_n to the negative gradient of f_{n-1}, and then takes a step γ so that $f_n = f_{n-1} + \gamma h_n$ minimizes the loss $L(y, f_n(\mathbf{x}))$. While AdaBoost identifies the failure points of weak learners by up-weighting them in subsequent iterations, gradient boosting uses the gradients of the loss function at each iteration to discover these failure points. Gradient boosting is an inherently more flexible framework compared to AdaBoost, but suffers from an inherent lack of explainability.

❖ Neural Networks

Neural networks represent a class of learning methods that automatically derive features as linear combinations of the inputs, and then model the target as a nonlinear function of those inputs.

The simplest neural network is the single layer perceptron, also known as a **feed-forward neural network**. This is not to be confused with Rosenblatt's perceptron algorithm earlier. In a single-layer FNN, we accept an $n \times p$-dimensional input $\mathbf{X}$ and aim to retrieve 1 output for regression and K outputs for classification. We then learn m **hidden states** $\mathbf{Z}_m$ as linear combinations of the features of $\mathbf{X}$. We additionally apply some non-linear transform to the result, which gives us additional flexibility in the form of nonlinearity. Our final output is then a linear combination of the hidden units, combined with another output function that morphs the result into something interpretable.

$$\mathbf{Z}_m = \sigma(\alpha^\top \mathbf{X}),\ \mathbf{T}_k = \beta_m^\top \mathbf{Z},\ f_k(\mathbf{X}) = g_k(\mathbf{T}).$$

Here, the sigmoid function $\sigma(\cdot)$ is typically used (for similar reasons to the logistic regression case). If we use the Gaussian radial basis function as the activation we call this a **radial basis function network**. Although it is hard to see from the functional form, we typically add a constant 1 as an input feature to allow learning a **bias** term at each node.

The output function is typically the **softmax** function

$$g_k(\mathbf{T}) = \frac{e^{\mathbf{T}_k}}{\sum_\ell e^{\mathbf{T}_\ell}}.$$

Simply put, the softmax transforms the outputs into a probability distribution over all the output classes. You can think of the $\mathbf{Z}_m$ as a basis expansion of the original inputs, at which point the neural network is simply a standard linear model. The important difference here is that the parameters of the basis functions are learned directly from the data.

In regression, we typically use a squared error as our loss function, and in classification we typically use cross-entropy. The classification output is $G(\mathbf{x}) = \arg\max_k f_k(\mathbf{x})$. The generic approach to minimizing the loss is via gradient descent, called **back-propagation** in the neural network setting. Since the overall functional form of the model can be thought of as a composition of additions, multiplications, and differentiable activations, we typically track the local gradients at each node by using the chain rule. In the **forward pass**, the current weights are used to evaluate $f_k(\mathbf{x}_i)$ over the training set; in the **backward pass**, the errors are computed and back-propagated through the network, and the weights are updated by taking a step (whose size is determined by the **learning rate** hyperparameter). The simple, local nature of backpropagation means that each hidden node's updates only depend on the the nodes it is directly linked to, meaning the computation becomes embarassingly parallel.

The dataset is typically never passed through the network all at once. Instead, the network will process **batches** of data. A single batch being passed through the network forward and backward is called an **iteration** or **step**. One full pass over the entire training set (consisting of $N/batch\_size$ iterations) is called an **epoch**.

Neural networks come with their own share of problems. Like with all other models, they have their own complexity parameter, captured in the number of layers and the number of nodes in each layer. They are prone to overfitting and are sensitive to their initialization constraints (for example, a network with all weights initialized to zero will never learn). Due to the long-tail asymptotes of the sigmoid function, gradients in the tail regions will be very close to zero, causing the network to stop learning (known as the **vanishing gradients problem**). The **rectified linear unit** (ReLU) activation function $f(x) = \mathbf{1}x > 0x$ solves this problem by having a derivative of 0 at negative values (inducing sparsity in the network) and 1 otherwise. Even though it might seem odd that having hard saturation for negative values (blocking backpropagation) would help the network, it does so in practice. In the cases where a neuron is only activated by negative values, we might want to use "Leaky" ReLU, letting a small gradient propagate through negative values. The opposite problem is when the gradients are large (>1) in large networks, causing the gradients to explode.

❖ Support Vector Machines

We've previously seen how separating hyperplanes can classify linearly separable classes. Support vector machines cover the case where the classes do overlap by finding linear decision boundaries in a transformed feature space (where the transformation may be nonlinear). For a dataset $(\mathbf{X}, \mathbf{y})$ where $y_i \in \{-1, 1\}$, we can define a classifier $G(\mathbf{x}) = \textbf{sign}(\mathbf{x}^\top \beta)$. The set of points such that $\mathbf{x}^\top \beta = 0$ is the separating hyperplane. We typically try to find a **max margin** classifier – i.e. a hyperplane that maximizes the distance between the two classes. In other words, we maximize M such that $y_i(\mathbf{x}^\top \beta) \geq M$ for $\|\beta\| = 1$ (an equivalent reformulation is to minimize $\|\beta\|$ such that $y_i(\mathbf{x}^\top \beta) \geq 1$ – then $M = 1/\|\beta\|$). The band separating the two classes is the **margin**. This case, where the classes are fully separable, is also known as a **hard margin SVM**.

If the data is not linearly separable, we might allow the classifier to make some number of mistakes. To do this, we can define a **slack variable** ξ for each data point. ξ_i is proportional to how far $\mathbf{x}_i^\top \beta$ is on the wrong side of the margin. We relax the optimization to then be $y_i(\mathbf{x}_i^\top \beta) \geq 1 - \xi_i$. A misclassification is then when $\xi_i > 1$ (this means it is on the wrong side of the classification hyperplane). We can bound $\sum \xi_i < K$ to limit us to up to K misclassifications. This is a **soft-margin SVM**, and is typically computed as a quadratic programming problem.

Soft-Margin SVM

$$\min \|\beta\| \text{ subject to } \begin{cases} y_i(\mathbf{x}_i^\top \beta) \geq 1 - \xi_i, \\ \xi_i \geq 0, \sum \xi_i < K \end{cases}.$$

The SVM classifier can be characterized entirely by the "boundary" points that lie exactly on the margin. In fact, the decision boundary is not at all impacted by points that lie well inside their own class boundary, meaning it is very easy to store and retrieve the SVM classifier. These particular points that define the margin are known as the **support vectors** (hence the name).

To solve the quadratic programming problem, we can use Lagrange multipliers. We can reformulate our previous soft-margin SVM formulation into a single Lagrange primal

$$L_P = \frac{1}{2}\|\beta\|^2 + C\sum \xi_i - \sum \alpha_i[y_i(\mathbf{x}^\top\beta) - (1-\xi_i)] - \sum \mu_i\xi_i.$$

Here, we use C instead of the constant K – the larger the value of C, the more we penalize misclassification. We take all our constraints and reformulate them in terms of the lagrange multipliers α_i and μ_i. The Lagrange dual is then

$$L_D = \sum \alpha_i - \frac{1}{2}\sum \alpha_i\alpha_{i'}y_iy_{i'}\mathbf{x}_i^\top\mathbf{x}_{i'}$$

Maximizing the dual is easier than minimizing the primal, and doing so gives us the appropriate $\hat{\beta}$ that we are searching for (details omitted).

The idea behind SVMs is that we can lift the data into a higher, possibly infinite-dimensional space using basis functions and then find a linear classifier in that transformed space. The key thing to note is the $\mathbf{x}_i^\top\mathbf{x}_i$ in the dual formulation. For basis functions h, this instead is the inner product $\langle h(\mathbf{x}), h(\mathbf{x}')\rangle$. This means that we don't really care about h itself, but only the inner product, the kernel $K(\mathbf{x},\mathbf{x}') = \langle h(\mathbf{x}), h(\mathbf{x}')\rangle$. This idea, where we can use an easy-to-evaluate kernel instead of ever having to process the possibly infinite set of basis functions, is often referred to as the **kernel trick**.

❖ Unsupervised Learning

So far we have assumed that we can train a model given the joint values of a training set consisting of $(\mathbf{x}, y)$ pairs. In this way, if we consider the training set to be drawn from a joint density $P(\mathbf{X},\mathbf{Y})$, then supervised learning amounts to estimate the conditional density $P(\mathbf{Y}|\mathbf{X})$. In an unsupervised setting, we instead try to learn $P(\mathbf{X})$ directly. While supervised learning algorithms typically focus on finding the optimum parameters θ to minimize a loss $L(\mathbf{Y},\theta)$, and likewise use that loss to score model performance, unsupervised algorithms have no such equivalent. Unsupervised algorithms might need to learn properties of $\mathbf{X}$ that are significantly more complicated, and must do so without any rigorous evaluation.r

11.1 Association Mining

Association mining attempts to find relationships between seemingly independent variables, for example *"If a customer buys eggs, there is an 80% chance they also buy bread."* This

particular example is known in finance as **market basket analysis**. An **association rule** consists of the **antecedent** ("If a customer buys eggs") followed by a **consequent** ("the customer buys bread"). The frequency that this pair appears in the dataset is called its **support**. The **Apriori algorithm** is a common association mining tool that is especially useful in optimizing complex, probabilistic database queries. The details are omitted here.

11.2 Cluster Analysis

Cluster analysis is the process of grouping objects into collections (**clusters**) such that objects inside the cluster are more closely related to each other than objects in another cluster. Clustering methods depend on a notion of similarity, which depends on the properties of the objects being clustered (much like a loss function in supervised settings).

Each data point in the dataset has p *attributes* (features). The dissimilarity between two datapoints is a weighted sum of some distance function $d(\cdot, \cdot)$ over the attributes (so $D(\mathbf{x}_i, \mathbf{x}_{i'}) = \sum d_j(\mathbf{x}_{ij}, \mathbf{x}_{i'j})$). Sometimes, as in the case for Euclidean distance, the d_j are the same for all features, but this is not always true. The Euclidean dissimilarity measure is the most popular, i.e. $d(x, y) = (x - y)^2$.

Combinatorial clustering algorithms *encode* observations into one of K predefined clusters by minmizing a "loss" function dependent on the pairwise dissimilarities between *every* pair of observations. Typically this is done through some kind of combinatorial optimization, such as by greedy descent – in many clustering scenarios, the feature space is too large to feasibly compute a global minimum, but a local minimum may be achieved.

11.2.1 K-means

K-means: Lloyd's Algorithm

1. For a given cluster assignment C, centroids (means) $\mathbf{m}_k$ are calculated to minimize the intra-cluster distances $\|\mathbf{x}_i - \mathbf{m}_k\|$.

$$\min_{C, \{\mathbf{m}_k\}} \sum_{k=1}^{k} N_k \sum_{C(i)=k} \|\mathbf{x}_i - \mathbf{m}_k\|^2$$

2. Given the current set of means, each observation is assigned to its closest cluster mean.

$$C(i) = \arg \min_{1 \le k \le K} \|\mathbf{x}_i - \mathbf{m}_k\|^2$$

3. (1) and (2) are repeated until the assignments no longer change.

Note that the above algorithm is very similar to the EM algorithm for Gaussian mixtures. In fact this is exactly the case, where in EM we assign "responsibilities" to each datapoint in the E-step, based on the mixture we believe it belongs to; and then in the M-step we compute the parameters of each mixture component based on the assignments in the E-step. In this way the EM algorithm can be thought of a "softer" version of K-means (since in K-means there is no probabilistic assignment – the clusters are deterministic).

K-means is often used in the data processing world for compression in the form of **vector quantization**. Suppose an 8-bit image; we can use K-means to reduce the number of colors in the image. Here the distance function is the distance between greyscale color values. If we can decrease the 8-bit image into 4 colors, we only need 2 bits per pixel, a huge improvement in efficiency. This form of compression is inherently **lossy**, meaning we cannot recover the original 8-bit values from the compressed version.

K-means refers to the particular problem setting where we are able to compute the pairwise Euclidean distance between any two pairs of points. In places where an obvious metric is not easily defined, we may instead to choose one of the datapoints (an **exemplar**) as the centroid – this is known as the K**-medoids** problem. This new restriction that the centroids must be datapoints themselves makes this problem drastically more computationally expensive. In practice, we may pre-compute all the pairwise dissimilarities in this case, saving it as a **dissimilarity** or **proximity matrix**, since there are no new distances that would need to be computed. K-means style algorithms still require intuition to select the number K, and selecting such a number dynamically is difficult.

11.3 Hierarchical Clustering

While K-means requires knowledge of the number of clusters K, other methods might either start from 1 cluster and recursively split that cluster until some performance threshold is guaranteed. Alternatively, they might start with many clusters (as many as N) and iteratively combine clusters until that threshold is achieved. The former method is known as **divisive** clustering and the latter is known as **agglomerative clustering**, both examples of **hierarchical clustering**.

Agglomerative clustering algorithms begin with each element representing a singleton cluster. At each step, the clusters that are closest together are merged together, decreasing the total number of clusters by one, until the desired number of clusters is reached. The dissimilarity between two groups is the minimum pairwise distance between two points in different clusters (this is known as **single-linkage** agglomerative clustering). Alternatively, **complete linkage** agglomerative clustering takes the maximum pairwise distance between two points in different clusters. Because single-linkage clustering only takes into account the closest two points, it can result in clusters with very large combinatorial diameters. Complete linkage faces the opposite problem, where sometimes

points will be closer to other clusters than to members of their own cluster. **Group average clustering** attempts to alleviate these problems by taking the average pairwise dissilmiarity between two groups to determine the intercluster dissimliarity.

A similar approach can be taken for divisive clustering, although divisive clustering is much less well-studied and is less popular than agglomerative clustering.

11.4 Principal Components

As seen earlier, principal components reveal the mechanism behind ridge regression. Principal components are projections of the data that provide a sequence of best approximations to that data. Consider a zero-mean, rank p dataset. The rank $q \leq p$ linear approximation can be given by the linear equation $f(\lambda) = \mathbf{V}_q \lambda$, where $\mathbf{V}_q$ is a rank-q orthonormal matrix. Then minimizing the reconstruction error amounts to minimizing $\sum \|\mathbf{x}_i - \mathbf{V}_q \lambda_i\|_2^2$, which can be done with $\lambda_i = \mathbf{V}_q^\top \mathbf{x}_i$, yielding the objective $\min_{\mathbf{V}_q} \sum \|\mathbf{x}_i - \mathbf{V}_q \mathbf{V}_q^\top \mathbf{x}_i\|_2^2$. The matrix $\mathbf{V}_q \mathbf{V}_q^\top$ acts as a projection matrix, and $\mathbf{V}_q \mathbf{V}_q^\top \mathbf{x}_i$ is the orthogonal projection of $\mathbf{x}_i$ onto the subspace spanned by the columns of $\mathbf{V}_q$. The solution to this problem comes in the form of the singular value decomposition $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where $\mathbf{V}_q$ is the first q columns of $\mathbf{V}$. The columns of $\mathbf{U}\mathbf{\Sigma}$ are the λ_i, the **principal components** of $\mathbf{X}$. It can be thought of that for a given point $\mathbf{x}_i$, $\mathbf{v}_j$ is the direction of axis j, and $\mathbf{u}_{ij}\sigma_j$ is the distance of the projected $\mathbf{x}_i$ from the origin. The largest principal components represent the linear projection of maximum variance. This variance-maximization makes principal component analysis (PCA) a powerful tool for separating, identifying, and visualizing clusters.

11.5 Spectral Clustering

Clustering methods like K-means struggle when the clusters in question are non-convex. To solve this problem, we model the data as a graph based on the pairwise similarity matrix $\mathbf{S}$. The vertices of the graph are the data points, and edges are drawn between points where the similarity is above some threshold. The weight of each edge is the similarity between those two points. Then clustering becomes a graph-partition problem, where intra-cluster nodes have high-weight edges and inter-cluster nodes have low-weight edges. The matrix of edge weights $\mathbf{W}$, known as the **adjacency matrix**, models the actual edges in the graph. If we subtract the *degree* of each node from the diagonal of $\mathbf{W}$ (the degree is the sum of all weights connected to that point), we end up with the **unnormalized graph Laplacian** matrix $\mathbf{L}$. Spectral clustering then finds the m eigenvectors $\mathbf{Z}$ corresponding to the m smallest eigenvectors of $\mathbf{L}$. We can then use K-means on the values in the rows of $\mathbf{Z}$ to yield a clustering of the data (so if, for the 2nd eigenvector, the first 10 values are < 0 and the next 10 values are > 0, we might then end up with two clusters, where the points in the clusters correspond to the indices in the eigenvector). In particular, the Laplacian matrix always has a zero eigenvalue with

constant eigenvector, so we are actually most interested in the 2nd smallest eigenvalue onwards.

11.6 Non-negative Matrix Factorization

Non-negative matrix factorization (NMF) is an alternative to PCA, where the data is assumed non-negative (such as in image data). The data matrix $\mathbf{X}$ is decomposed into $\mathbf{WH}$. These matrices are determined by maximizing the log likelihood of a model where the data is assumed to be Poisson with mean $\mathbf{X}$:

$$L(\mathbf{W}, \mathbf{H}) = \sum_{i=1}^{N} \sum_{j=1}^{p} \mathbf{x}_{ij} \log(\mathbf{WH})_{ij} - (\mathbf{WH})_{ij}.$$

This form admits no closed-form solution, and as such must be approximated numerically. The low-dimensional basis yielded by NMF are more interpretable than that of PCA, but NMF suffers from non-uniqueness, as multiple choices of basis vectors may yield feasible solutions.

❖ Random Forests

Bagging methods reduce variance of a predictor by ensembling multiple learners together. For regression, we can average an ensemble of weak regressors, and for classification we can vote by committee, where each learner is trained on a bootstrap sample. Boosting methods also propose a committee approach, except the committee of weak learners evolves over time, an each learns a specific weight. Boosting almost universally outperforms bagging.

Random forests are a modification of bagging that averages a collection of *decorrelated* trees. Random forests perform similarly to boosting, and are widely used due to ease of implementation.

Trees are generally ideal candidates for bagging, since they have low bias when sufficiently deep. Since the trees are typically i.i.d., the mean of the average is equivalent to the mean of any one tree, so the bias of the model stays stable. But the variance will be reduced by a factor of the number of trees. However, this is not true if the trees are not independent. The more correlated the trees are, the less beneficial averaging becomes. Random forests address this issue by attempting to reduce the correlation of trees as much as possible without contributing to variance. This is done by, for each tree, selecting a subset of predictor variables as candidates for splitting at each node. Typically the number of variables chosen is around $\sqrt{p}$ where p is the total number of variables.

Random forests perform poorly when only a small percentage of variables are relevant, since at each split there is a low chance an important variable will be selected. Addi-

tionally, while intuitively random forests should not be able to overfit the data (since they keep the variance so far down), in practice an average of many fully-grown trees can result in too complex of a model and cause even minute differences in variance to explode.

❖ Graphical Models

A graph is a set of vertices and edges joining pairs of vertices. A graphical model is a way to represent the joint distribution of a set of random variables, where the nodes are each random variable and the edges represent some conditional dependence. Graphical models where the edges have no direction are called **Markov random fields** (MRFs). The edges of a graph are parameterized by **potentials**, which describe the strength of the conditional dependence between the two corresponding vertices. Much of traditional algorithmic graph theory applies here, and the details are left to a more thorough treatment of classical algorithms.

If two vertices are not connected with an edge, that means they are conditionally independent given the rest of the graph. A graph can be partitioned into subgraphs. If a subgraph **separates** two other subgraphs, meaning every path from subgraph A to B passes through subgraph C, then all of A is conditionally independent of B given C. These are known as the pairwise and global Markov properties of the graph, respectively.

The global Markov property allows for a decomposition of the graph into **cliques**, which are complete (fully-connected) sugraphs. A **maximal clique** is one where no additional vertices may be added while maintaining this property. Then each clique is assigned a specific **potential** function that captures the dependence present within in that clique. The density function over $\mathcal{G}$ is the product of all clique potentials, so for cliques C, and potential φ,

$$f(x) = \frac{1}{Z} \prod_{C \in C} \varphi_C(x_C).$$

The property that a graph has "independent" cliques is known as the **Hammersley-Clifford theorem**. Graph estimation methods often decompose these graphs into maximal cliques.

Probabilistic graphical models comprise their own complete field of study, and as such the details here are left to a more thorough exploration, such as in Koller and Friedman's *Probabilistic Graphical Models*.